REMARKS

1. Figure 1 is objected to due to reference numbers 14, 126 and 148, which are shown on the Figure but omitted in the specification. Applicant gratefully acknowledges the Examiner's reading of the specification, and herein amends page 7 of the specification to include the reference numbers.

2. Claim 7 stands rejected under 35 USC 112. It is asserted that there is insufficient antecedent for "said computer software" in line 3 of the claim. Applicant notes that the preamble of claim 7, on lines 1 and 2, states "A computer-implemented apparatus for detecting one or more zombie global breakpoints for debugging computer software . . . ," which provides an antecedent basis for the term "computer software" in line 3. Applicant nevertheless understands that the Examiner may consider the claims to be in a more readable style if terms subsequent to the preamble are reintroduced and may consider that this style for claim 7 is more consistent with that of the other claims in the application. Applicant considers that in this particular case the suggested change does no harm to the claim's meaning, thanks the Examiner for considering readability, and responsively amends the claim herein as requested.

3. Claims 1-18 stand rejected under 35 103(a) as being unpatentable over Alverson et al. (6,480,818) and further in view of Kakivaya et al. (6,546,443). Applicant respectfully disagrees.

It is conventional that when a process encounters a breakpoint a debugger temporarily removes the breakpoint. An instruction for which the breakpoint was originally substituted is then executed, and then the breakpoint is immediately replaced so that if the instruction is again encountered the breakpoint will fire. The present invention deals with a problem that arises when two threads encounter the same breakpoint and the debugger temporarily removes the breakpoint for one of the threads while processing of the breakpoint for the second thread is still pending. The problem is that when the processing of the breakpoint occurs for the second thread the breakpoint may be absent because it has been temporarily removed for the first thread. The absent breakpoint is referred to in the present application as a "zombie" breakpoint. The above described problem will be referred to herein as the "zombie breakpoint problem."

While Applicant is certain the specification and claims are understandable to a person of ordinary skill in the art, upon review Applicant recognizes that certain claim language in the present application is arguably a bit awkward. That is, the original claim 1 stated that the method includes "checking . . . to determine if a breakpoint . . . is at an address; if . . . breakpoint cannot be determined at said address, verifying if a breakpoint . . . continues to exist at the address where the breakpoint fired . . ." The claim might be possibly mis-read in a such way as to give rise to the question, "if the breakpoint cannot be determined at said address then how can it be verified if the breakpoint continues to exist at the address?" Consequently, Applicant herein submits amendments to more certainly ensure the avoidance of such a misunderstanding.

Specifically, claim 1, for example, is amended to state that the checking of the breakpoint data structure is to determine if the *data structure has an entry* for a breakpoint known to a debugging process for a certain address where a breakpoint fired. Then, the claim goes on, there is a step of verifying if a breakpoint condition continues to exist at the address where the breakpoint fired if no entry is found by the step of checking the data structure for the known breakpoint. Thus the amendment makes more clear that in the first step the checking looks in one place, the data structure, and in the second step the verifying looks somewhere else, which could be the actual memory address for the breakpoint or possibly in a breakpoint register. Independent claims 7 and 13 are likewise amended. No new matter is added by the amendments because support for the amendments is found in the original specification at page 3, lines 10 through 13 and page 6, lines 10 through 12.

Much of what the Office action cites from Alverson is relevant to the present invention in the sense that Alverson is also concerned with debugging for multithreaded processing, but the relevance does not extend to Alverson teaching the same method and structure for solving the zombie breakpoint problem. Alverson, like the present invention, concerns a debugger for controlling execution of target code for which there are multiple thread processes. Page 9, lines 15 through 17. Alverson recognizes and deals with the same zombie breakpoint problem as the present invention. Page 13, lines 3 through 10. However, Alverson handles this problem in a different manner, and thus not only fails to suggest the present claimed invention, but actually teaches away from the present invention.

Applicant contends that much of the teaching by Alverson relied upon in the rejection of the Office action does not provide any particular insight into and does not suggest the teaching of the present patent application. Applicant has considered the teaching of Alverson referred to by the Office action, and offers the following by way of summary in order to place the cited teaching into context.

Alverson advocates and teaches about a debugger that has a root nub and individual nubs for the respective target thread process. Page 9, lines 5 through 6. The debugger nubs obtain state information about their respective target thread processes. Page 9, lines 28 through 30. When an exception occurs, a general trap handler determines the type of execution causing the trap so that the general trap handler can invoke the right exception trap handler. Page 10, lines 37 through 39. If the trap is caused by a breakpoint then a breakpoint exception handler can be invoked. The breakpoint exception handler can interact with the debugger nub thread for the target thread process that gave rise to firing of the breakpoint, in order to allow debugger control. Page 10, lines 44 through 54. The debugger nub can get information about the target thread from a save area. Id.

Alverson indicates that it is known to use a domain signal to manipulate all the threads in a system. Page 11, lines 31 through 33. Alverson teaches that the debugger nubs may need to ignore the domain signal. Page 11, lines 33 through 35. Also, a target thread process may need to temporarily ignore the domain signal while its nub accesses the target's data structure. Page 11, lines 37 through 41.

More to the point of the present invention, Alverson offers a solution to the zombie breakpoint problem, as follows. When a breakpoint is inserted for an instruction, the debugger generates an out of line instruction emulation group so that the instruction for which the breakpoint is substituted can be executed in another area of memory, that is, "out of line. " Page 13, lines 11 through 17; see also FIG's 4A and 4B. Instead of temporarily replacing the instruction back in line after encountering the breakpoint, as is conventionally done, the breakpoint handler transfers execution to the instruction emulation group. Page 13, lines 43 through 48. After executing the instruction in the instruction emulation group, that is, after executing the instruction for which the breakpoint was substituted, the target thread resumes execution with the next instruction in the original, in-line code. Page 14, lines 14 through 17. Thus, by generating the out of line instruction emulation group and executing the substituted instruction in the instruction emulation group instead of temporarily replacing the instruction back in-line, "the processing of the breakpoint has been performed *without removing the BREAK* *instruction from the target code instructions*. Thus if another target thread had executed the same instructions while the breakpoint for the first target thread was being processed, the second target thread will also encounter the breakpoint instead of inadvertently missing a temporarily absent BREAK instruction." Page 14, lines 31 through 38 (emphasis added).

This teaching by Alverson is in direct contrast to that of the present invention. According to the present invention an aspect of the conventional arrangement is maintained, according to which upon encountering a breakpoint the breakpoint is *temporarily removed* from the line of code so that the instruction for which the breakpoint was substituted can be replaced and executed. That the breakpoint is temporarily removed is clear from the language of the application, which states that the breakpoint cannot be found, indicating the breakpoint has been removed. Page 5, lines 24 through 30 ("In the embodiments of the invention, breakpoint-discrimination logic in a debugger is responsible for looking up breakpoint data structures to determine if there is a breakpoint known to the debugger at the address where the breakpoint is fired. If this routine cannot find a known breakpoint, the debugger recognises that this may be a zombie breakpoint and does just one more check: verify if the breakpoint condition is still at the address where the breakpoint is fired. If the breakpoint is not there, a zombie breakpoint is identified and handled accordingly."). The application also explicitly states that breakpoint removal logic "lifts," i.e., removes, the breakpoint. Page 6, lines 10 through 12 ("For this approach to work smoothly, the breakpoint removal logic takes care to first lift the physical breakpoint instruction from the breakpoint location before removing the breakpoint entry from the data structures of the debugging.").

That the breakpoint is removed is also made clear in the following passage:

> The following description of the preferred embodiment is from the point of view of the debugger/tool utilising the solution described above:
> a. Apply global breakpoints by inserting a special breakpoint instruction (INT3) at the desired locations.
> b. Assuming two threads, executing on different processors, hit the same INT3 at the same time (events A and B).
> c. Assume Event A enters the breakpoint discrimination logic first.
> d. Event A is determined to be a breakpoint and is reported (handle Event A).
> e. User *removes* the breakpoint being handled (INT3) while handling Event A.
> f. Next, Event B enters the breakpoint discrimination logic.
> g. Event B is not considered a breakpoint as the breakpoint has been removed in step e above.
> h. Verify if there is an INT3 instruction at the trapping location.
> i. The INT3 Instruction is not there as step e above *removed* the breakpoint.

> Thus, event B is due to a *breakpoint that has been removed after being hit* (i.e., a zombie). Once a zombie breakpoint is detected, the breakpoint can be handled according to a policy defined by the debugger to handle zombie breakpoints.
> Page 6, line 16 through page 7, line 1 (emphasis added).

The independent claims in the present invention particularly point out this distinctive feature of the invention. For example, claim 1 states that "if said breakpoint does not exist," i.e., because it has been temporarily removed, the breakpoint is identified as a zombie breakpoint. This is contrasted to the conventional result, according to which it is incorrectly concluded, due to the absence of the removed breakpoint, that the exception was not caused by a breakpoint. Page 5, lines 18 through 20 (". . . the operating system incorrectly assumes the breakpoint exception to be an unhandled exception and takes the default action, which usually results in the debuggee terminating.").

The Office action also relies on Kakivaya, which deals with synchronization issues arising from multithreading, but does not concern breakpoint handling by a debugging program. With regard to debugging, Applicant is unable to find more than a passing reference by Kakivaya to the fact that in a multithreading environment it is harder to debug a program. Applicant is unable to find even the mere mention by Kakivaya of the term "breakpoint." Certainly, Kakivaya does not suggest the teaching that is absent from, and even incongruous with, Alverson regarding dealing with an absent breakpoint by checking a breakpoint data structure to determine if the data structure has an entry for a breakpoint known to a debugging process for a certain address where a breakpoint fired; verifying whether a breakpoint condition continues to exist at the address (i.e., verifying whether the breakpoint has been removed) if no entry is found by the checking for the known breakpoint; and identifying the breakpoint as a zombie breakpoint if the breakpoint does not exist, i.e., because it has been temporarily removed.

In addition to the above discussed independent claims 1, 7 and 13, the present application has a number of dependent claims, 2-6, 8-12 and 14-18. Since the broadest, independent claims are patentably distinct, the dependent claims are likewise patentably distinct merely based on their dependence upon the respective independent claims. Moreover, the dependent claims 2-6, 8-12 and 14-18 are all the more patentably distinct due to their respective additional limitations.

Docket JP920000280US1

Appl. No.: 09/732,250
Filed: December 7, 2000

PRIOR ART OF RECORD

Applicant has reviewed the prior art of record cited by but not relied upon by Examiner, and assert that the invention is patentably distinct.

REQUESTED ACTION

Applicant contends that the invention as claimed in accordance with amendments submitted herein is patentably distinct, and hereby requests that Examiner grant allowance and prompt passage of the application to issuance.

Respectfully submitted,

Anthony VS England

Anthony V. S. England
Attorney for Applicants
Registration No. 35,129
512-477-7165
a@aengland.com